Synopsis: swapinit: proc (fixed) returns (fixed);

Usage: old_sector = swapinit (new_sector);

where NEW_SECTOR is the sector address in external memory to set up as the start of the swap file. SWAPINIT reinitializes the swapping mechanism (i.e., resets all the swapping variables), and returns the sector address that was previously the start of the swap file.

SWAPINIT can be used simply to reinitialize the swapping mechanism by using it recursively, as in the second example below. SWAPINIT is called with any value in order to obtain the location of the current swap file, and then SWAPINIT is called again with this returned sector number. The swapping mechanism is initialized, but the swap file is located in the same place it was before.

Note that SWAPINIT only sets a pointer to the swap file, it does not actually put the swap file into external memory. The swap file is normally loaded into external memory whenever a program is loaded into memory by the system.

Example:

```
call swapinit (256); /* swapping starts at sector 256 */

call swapinit (swapinit (0)); /* reinitialize swapping
                                 mechanism */
```

TAN — computes tangent — ARITHMETIC

Synopsis: tan: proc (floating) returns (floating);

Usage: result = tan (num);

TAN returns the tangent of NUM, where NUM is the angle in radians. Tangent is equivalent to SIN/COS.

Example:

```
dcl (x, y) floating;

x = tan (y);
```

See also: ATN
SIN
COS

WRITE — writes a word to an interface device — HARDWARE

Usage: write (device_number) = value;

The WRITE statement writes a VALUE to the interface device specified by DEVICE_NUMBER. DEVICE_NUMBER can be a constant expression or a variable expression, although the WRITE will be much slower in the latter case.

If an attempt is made to read a device that is not in the system, the computer will halt.

Example:

```
dcl DAC literally '"66"';
dcl i   fixed;

write (DAC) = i;
```

See also: READ

Synopsis: writedata: proc (fixed, fixed, fixed array, fixed);

Usage: call writedata (ms_sector, ls_sector, buffer, length);

where MS_SECTOR and LS_SECTOR form a 32-bit word pair that identifies the device and sector number that is to be written. The upper eight bits (byte) of this word pair specify the storage device number and the lower 24-bits specify the sector number on that device. LENGTH words of data will be written from BUFFER to this device and sector location.

When using WRITEDATA to access Winchester systems with more than one drive attached to a device, XPL will automatically determine which physical Winchester disk contains the specified logical sector.

NOTE: If the device you are using with WRITEDATA is a SCSI device, you must insert either :-XPL:SCSI or :-XPL:SCSISWAP into your program.

Example:

```
dcl buf (1024) floating;

call writedata ((shl (7, 8) or 4), 845, buf, 512);
```

In this example, the first 512 words of BUF are written to the W1 Winchester disk (upper byte = 7), starting at sector 262,989 (lower 24 bits = 4*65,536 + 845). WRITEDATA automatically maps this logical sector number to the correct physical Winchester disk on the device.

See also: READDATA

Appendix B - Compilation Control

Compile-time Options

The Scientific XPL compiler provides two compile-time switches that are used to provide information about compilations. If the sequence $D (or $d) is found in any comment in the program, statistics about the compilation are "dumped" (hence the D in $D) to the current output device. This works for modules as well as for the main body of code, and works for both the COMPILE and RUN commands. The current output device is normally the terminal, but can be redirected to the printer by suffixing RUN or COMPILE with ',P' as follows:

```
run,p          sends output from the RUN to the printer
compile,p      sends output from the COMPILE to the printer
```

The other compile-time switch is $M. If the sequence $M (or $m) is found in any comment in the program, a symbol table is output after the link process and a sequence number table is output as the compilation completes. The $M switch produces no output for programs that are RUN or for modules.

The symbol table consists of each library that was linked into the final program in the order it was linked, as well as its starting address in memory, the start of its string and constants area, and the start of its variable or RAM area. Thus, it's a sort of link map (hence the M in $M). In addition, information about each PUBLIC variable defined in that library is output in alphabetic order by symbol name.

The sequence table consists of a list of line numbers (for each library) with the octal address of the object code created for the statement found at each line number.

It is possible to redirect the symbol table and the sequence table to a file. In order to do this you must save a file named -SYMTAB- in the catalog where you perform your compilations as follows:

```
new -SYMTAB-
save, <sectors>
```

where <sectors> is the number of sectors you wish to reserve for the symbol table and sequence table output. Ten (10) or twenty (20) is a good number to start with. If -SYMTAB- is not large enough, the output will be redirected to the current output device (the terminal or the printer). If you still want the tables output to a file, you can create a longer -SYMTAB- file and try compiling your program again. The compiler automatically redirects these two tables to the file -SYMTAB- if it exists in the current catalog.

The CONFIGURATION Statement

The CONFIGURATION statement allows the programmer to specify certain system configuration restraints that will be followed during the compilation of a program. The statement takes the form:

```
configuration <symbol>, <symbol>, <symbol>, ...;
```

where any number of <symbol>s selected from the list below can appear:

SMINI	system device is superfloppy diskette
DMINI	system device is double-density diskette
MINI	system device is single-density diskette
MAXI	system device is maxi (8") diskette
MODELB	CPU Model-B
MODELC	CPU Model-C
MODELD	CPU Model-D
MULDIV	use hardware Multiply/Divide unit
NOMULDIV	use software Multiply/Divide routines
MEMORY <size>	absolute memory size (in words)
PTYPE <#>	set terminal type to this number (for PRINT statement)
STYPE <#>	set printer type to this number (for SEND statement)

Terminal and printer type literals are defined in the file SYSLITS in the -XPL programming library (:-XPL:SYSLITS).

For example:

```
configuration dmini, modelB, nomuldiv, memory 40*1024,
              ptype t#hardcopy;
```

The information provided in the CONFIGURATION statement becomes the default configuration stored in the configuration table of the program. Without a CONFIGURATION statement, the configuration of the program will be the same as the system that it was compiled on (the configuration of that system's Monitor). For more information on the configuration table, see the appendix "Memory Layout".

RAM and PDL Statements

The XPL compiler has the capability to compile programs that will run on a ROM (read only memory) based machine. The RAM statement is used to tell the compiler where the random access memory is located in the computer system memory configuration. The keyword RAM must be followed by a fixed point constant expression, which is the starting address of random access memory:

```
ram <start>;
```

An example of the RAM statement is:

```
ram "1400";
```

If no RAM statement appears, the RAM is assumed to start at the lesser of 8192 or the end of the program's object code.

The XPL compiler uses a Push Down List (PDL) while performing a procedure call. The push down list is used to store information such as automatic variables and return locations during procedure calls and interrupt processing. The amount of memory reserved for the PDL may be changed with the PDL statement. A default length of 256 words is used for the push down list if the program contains no PDL statement. If a program needs more storage for the push down list, as in the case of recursive procedure calls or recursive interrupt processing, its length can be increased using the PDL statement. The keyword PDL is followed by a fixed point expression that sets the word length of the push down list:

```
pdl <size>;
```

An example of to increase the size of the push down list is shown below:

```
pdl 1024;     /* set up PDL of 1024 words */
```

In no case can the push down list length be set to zero, or undefined program operation will occur. There is no limit (other than the amount of available memory) on the maximum size of the PDL.

EOF Symbol

The EOF symbol signals the compiler that it is the end of the file. This is used mostly as a debugging aid, for compiling and running a portion of a program. EOF can occur immediately before or immediately after any program statement.

Appendix C - Summary of XPL

Comments and White Space

Comments and white space (spaces and blank lines) can appear anywhere in an XPL program. They are totally ignored by the compiler. Comments start with /* and end with */. Comments cannot be nested (i.e., the sequence /* cannot appear within a comment).

XPL Data Types

XPL supports two basic data types:

fixed:	signed fixed point numbers in the range of -32768 to +32767; can be interpreted as unsigned numbers from 0 to 65535.
floating:	floating point numbers in range of 5.5 E-20 to 9.0 E+18.

From fixed point numbers are derived:

boolean:	logic value true or false.
pointer:	address of another variable or pointer into memory or null.

Constants

XPL supports three fixed point constants: decimal, octal, and hexadecimal. XPL also supports floating point constants. Examples are shown below.

Decimal	Octal	Hexadecimal	Floating
256	"17"	"H0001"	0.001
-142	"00276"	"HFFFF"	-1.04
65530	"377"	"HA2C"	124.675

XPL supports two boolean constants: true and false.

XPL supports one pointer constant: null. All unsigned fixed point numbers are also valid pointer constants.

XPL supports string constants. These are specified by a sequence of up to 128 characters enclosed in apostrophes. An apostrophe within a string constant is specified with two consecutive apostrophes. Examples are 'Hello there.' and 'Don''t touch that dial!'.

Identifiers

Identifiers are used throughout XPL to specify variable names, procedure names, label names, macro names, and module names. An identifier is a sequence of up to 32 letters, digits, and special characters that begins with either a letter or one of the special characters (except period). The special characters are number sign (#), dollar sign ($), underscore (_), and period (.). There is no distinction made between uppercase and lowercase letters. Examples are f#ms_sector, term_idle, answer$, and _sys11B_flag.

Storage Classes

XPL supports four storage classes for variables: automatic, static, public, and external. They are differentiated by scope, lifetime, and when initialized (all XPL variables are initialized to zero). When no storage class is specified, it is assumed to be static (except in recursive procedures where it is assumed to be automatic).

Automatic Variables:

Scope:	Rest of block declared in.
Lifetime:	Life of procedure declared in.
Initialized:	At entrance to procedure declared in.

Static Variables:

Scope:	Rest of block declared in.
Lifetime:	Life of program.
Initialized:	At start of program execution.

Public Variables:

Scope:	Rest of block declared in. Entire program if referenced with EXTERNAL.
Lifetime:	Life of program.
Initialized:	At start of program execution.

External Variables:

Scope:	Rest of block declared in. Used to reference PUBLIC variables.
Lifetime:	No storage associated. Refers to corresponding PUBLIC variable's storage.
Initialized:	No initialization associated since no storage.

Expressions

Expressions are composed of operands (variables, constants, or function calls) combined with operators. The XPL operators are listed below from highest to lowest precedence; operators on the same line are of equal precedence. Operators of equal precedence are evaluated from right to left. Parentheses can be used to force a specific order of evaluation.

```
shr  shl  rot
not
*    /    mod  %   fdiv
+    -
=    ~=   <    <=  >    >=  ieq  ine  ilt  ile  igt  ige
and  or   xor
```

The following symbols are aliases for the operators on the left:

```
not      ~   ^
~=       ^=  <>
and      &
or       ¦   \
```

Arithmetic Operators:

+	Addition (binary) or Positive (unary)
-	Subtraction (binary) or Negative/Two's complement (unary)
*	Multiplication
/	Division
mod	Modulus (remainder)
%	Fractional multiply
fdiv	Fractional divide

Signed Relational Operators:

=	Equal to
~=	Not equal to
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to

The value of a relational operation is either true or false.

Unsigned Relational Operators:

ieq	Is equal to
ine	Is not equal to
ilt	Is less than
ile	Is less than or equal to
igt	Is greater than
ige	Is greater than or equal to

The value of a relational operation is either true or false.

Logical Operators:

not	Logical negation
and	Logical and
or	Logical or

The value of a logical operation is true or false. The XPL compiler will evaluate only as much of a logical expression as is necessary to determine if it is true or false. For example, if the first term of an AND expression is false, the rest of the expression will not be evaluated. It is therefore inadvisable to embed a function call that must always occur within a logical expression unless it is the first term; the XPL compiler guarantees to evaluate logical expressions from left to right.

Bit Operators:

not	One's complement
and	Bitwise and
or	Bitwise inclusive or
xor	Bitwise exclusive or
shr	Shift right
shl	Shift left
rot	Rotate left

Statements

XPL programs are constructed of a series of XPL statements. All XPL statements are free format (i.e., many statements can occur on one line or one statement can span multiple lines) and end with a semicolon (;). The simplest statement is the null statement which means do nothing and is specified by just a semicolon.

In the following discussion, "scalar variable" refers to both simple variables and subscripted variables (array elements).

Declaration Statement:

Before an XPL variable can be used, it's data type and storage class must be declared.

```
dcl <id> literally <string>;                          /* literal */
dcl <id> <type> <class>;                               /* scalar */
dcl <id> (<size>) <type> <class>;                      /* array */
dcl <id> <type> data <class> (<list>);                 /* data */
dcl <id> proc (<parms>) returns (<type>) <class>;      /* proc */
dcl <id> label <class>;                                /* label */
```

where <id> is an XPL identifier, <string> is a string constant, <type> is an XPL data type, <class> is an XPL storage class (optional), <list> is a comma-separated list of data items or a string constant, and <parms> is a comma-separated list of the data types of the procedure's formal parameters. For procedures, the <class>, if it appears, must be either recursive for a forward reference to a recursive procedure or external to reference a public procedure.

Assignment Statement:

```
<id> = <expression>;
```

where <id> is the identifier of a scalar variable that is assigned the value of <expression>.

Compound Statement:

```
do;                 /* compound statement */
   <stmts>;
end;
begin;              /* compound statement; begin new program block */
   <stmts>;
end;
```

where <stmts> is any number of XPL statements. These statements are used to treat a group of statements as one. Note that BEGIN only differs from DO in that BEGIN begins a new level of identifier localization.

Flow of Control:

```
  if <expr>                  /* IF statement */
  then <stmt>;               /* execute if <expr> is TRUE */
  else <stmt>;               /* execute if <expr> is FALSE (optional) */
```

where <expr> is a boolean expression and <stmt> is any XPL statement (including the compound statement). The ELSE clause of the IF statement is optional.

```
  do while (<expr>);         /* DO while */
     <stmts>;                /* execute while <expr> is TRUE */
  end;
```

where <expr> is a boolean expression and <stmts> is any number of XPL statements. The loop is not entered if <expr> is initially FALSE. <expr> is evaluated every time through the loop.

```
  do <id> = <expr1> to <expr2> by <expr3>; /* iterative DO */
     <stmts>;                /* execute while <id> <= <expr2> */
  end;
```

where <id> is the identifier of a scalar variable (except an array element), <expr1> is the lower bound of the loop (initial value of <id>), <expr2> is the upper bound of the loop (final value of <id>), and <expr3> is the step (the value to add to <id> each time through the loop). The loop is not entered if <expr2> < <expr1> initially, although <id> will be set to <expr1>. <expr2> - <expr1> must be less than 32768 for the loop to work correctly. Each of the three expressions is evaluated ONCE before entering the loop and never evaluated again. The value of <id> at the end of loop execution is indeterminate. The step is optional, but if present must evaluate to a positive value; if no step is provided, the step will be one. As a special case, a negative constant can be used as a step; the loop executes while <id> >= <expr2>.

```
  do case (<expr>);          /* DO case */
     <stmt0>;                /* execute if <expr> = 0 */
     <stmt1>;                /* execute if <expr> = 1 */
     ...
     <stmtN>;                /* execute if <expr> = N */
  end;
```

where <expr> is a fixed point expression and <stmt0> through <stmtN> are any XPL statements (including the compound statement). If <expr> igt N, no statements are executed.

```
  <id>: <stmt>;              /* program label */
  goto <label id>;           /* GOTO statement */
```

where <id> is an XPL identifier, <label id> is an XPL identifier that represents a label and <stmt> is any XPL statement.

Procedures:

```
<id>: proc (<parms>) returns (<type>) <options>; /* proc defn */
   <stmts>;
end <id>;
return (<expr>);                    /* return from procedure */
```

where <id> is an XPL identifier, <parms> is a comma-separated list of formal parameters, <type> is the type of value returned (e.g., fixed), <expr> is an expression of type <type>, <stmts> is any number of XPL statements, and <options> is any combination of the following (in this order): PUBLIC, RECURSIVE, and/or SWAP. The parameter list, the returns attribute, and the options are all optional. The expression in the RETURN statement must appear in procedures that return values (functions) and must NOT appear in procedures that do not, nor in WHEN statements. Formal parameters are automatic in recursive procedures, static in all others.

```
call <proc id> (<act parms>);     /* procedure call */
<id> = <proc id> (<act parms>);   /* function call  */
```

where <proc id> is an XPL identifier representing a procedure, <act parms> is a comma-separated list of actual parameters. The actual parameter list must match the formal parameters of the called procedure's definition in number and type.

Modules and Libraries:

```
module <id>;                  /* define a module */
   <stmts>;
end <id>;
```

where <id> is an XPL identifier and <stmts> is any number of XPL statements. In a file that defines a module, no XPL statements can appear outside the module.

```
insert '<treename>';              /* insert source file */
enter '<treename>';               /* change search catalog */
library '<treename>';             /* insert a library */
```

where <treename> is a valid treename. Libraries are executed before the main program and appear in the order in which the compiler finds the LIBRARY statements (nested libraries are linked as they are found).

Interrupts and Exceptions:

```
enable;                                 /* turn interrupts on */
disable;                                /* turn interrupts off */
when <interrupt id> then <stmt>;        /* WHEN statement */
invoke <interrupt id>;                  /* invoke a WHEN statement */
```

where <interrupt id> is one of the several predefined identifiers for interrupt conditions (e.g., TTIINT, D40INT, BREAK) and <stmt> is any XPL statement (including a compound statement). When a program begins execution, interrupts are disabled. The two special exception handlers WHEN BREAK and WHEN DISKERROR are always active and do not require the use of the ENABLE statement. Note that swapping procedures should NOT be called from within a WHEN statement. Interrupts generated by the D50 (TTIINT and TTOINT), the D03, the D136, and the D16 are automatically cleared by the XPL runtime system.

Terminal Input and Output:

```
input <input specifier>; /* get numeric input from user */
linput <string id>;      /* get textual input from user */
```

where <input specifier> is a comma-separated list of identifiers for scalar variables to be input into and <string id> is the identifier for a fixed point array (string) that has 65 elements (0-64; to hold up to 128 characters).

```
print <print specifier>;        /* print on terminal screen */
send  <print specifier>;        /* send to computer/printer */
```

where the <print specifier> is a comma-separated list of print expressions. A print expression can be any one of the following:

```
<expression>            /* to print a decimal number  */
<string constant>       /* to print a string constant */
string (<string id>)    /* to print a string variable */
octal (<expression>)    /* to print an octal number   */
chr (<expression>)      /* to print a character by ASCII value */
```

where <expression> is any XPL expression, <string constant> is a string constant, and <string id> is the identifier of an XPL format string (fixed point array). Both PRINT and SEND automatically terminate the output with a carriage return/linefeed pair; this can be disabled by following the last print expression with a comma.

Appendix D - Compiler Error Messages

The efficiency of Scientific XPL derives from the three pass architecture of the Scientific XPL Compiler. Pass 1 of the compiler reads through the source file, performs all the symbol table processing, and creates an intermediate file. Pass 2 creates another intermediate file containing all the instructions for the program. Pass 3 then links in all libraries, throws away dead code, and creates the final object file.

Because the compiler generates very different types of error messages for each pass, the compiler error messages are divided into three sections in this appendix: Pass 1, Pass 2, and Pass 3. The progress of the compiler can be followed by pressing any key on the terminal while a program is compiling. A status message will be printed telling which pass the compiler is currently in and, if appropriate, the line number and file that is currently being processed.

Types of Compiler Errors

Pass 1 will process the main program, as well as any source files that are included with the INSERT statement. Any inserted files are processed as the compiler finds INSERT statements. This pass is where most of the syntax errors will occur. Such things as undefined variables, argument type mismatches, and incorrect statement formats will be found during Pass 1.

Pass 2 has relatively few error messages. If the program is going to be too large for memory, or too many procedures or labels are declared, this pass will flag an error. The line numbers given in these error messages are not completely accurate; an intermediate file is being processed rather than the source file.

Pass 3 is when all libraries are linked into the main program, so these errors tend to concern problems with libraries that are included with LIBRARY statements. Pass 3 detects such things as external variables that do not have a corresponding public definition and libraries that were compiled with a different compiler or for a different processor.

Many messages have a line number and filename appended to them, as in this example:

```
Incorrect format at line 00023 in file PROG-1
```

The filename would only be included if the error occurred in an inserted file rather than the main program.

The System Work File .WORK

The compiler uses the system file .WORK, located in the top-level catalog of the system device, to store data as it compiles a program. If .WORK is missing or is not large enough, the compiler will flag an error. This system file should be of the file type DATA and it's length must be a multiple of 8 sectors.

IMPORTANT: If you change the .WORK file at any time, make sure to force a cold boot (type BOOT at the terminal) before doing anything else on your system.

Pass 1 Error Messages

Argument type does not match previous proc defn

> The parameter type list of the forward reference declaration for this procedure does not match the parameters of this procedure.

Argument types do not match

> The type of an actual parameter in this procedure call does not match the type of the corresponding formal parameter.

Cannot reference global automatic variable

> An attempt has been made to access an automatic variable that is declared within the procedure that contains this procedure. Change the global variable's storage class to STATIC or pass it as a parameter to this procedure.

Duplicate definition for '<identifier>'

> This identifier has already been declared at the current scope level.

Duplicate 'WHEN' statement

> A WHEN statement with the same interrupt identifier (e.g., d03int) as this WHEN statement appears elsewhere in the program. Only one WHEN statement for each interrupt identifier is allowed.

END label does not match

> The identifier following the END of a procedure or module definition is not the same as the identifier (name) for this procedure or module. This usually occurs when there are too many or too few END statements within a procedure or module.

Expression not allowed

A variable appears within an expression that must be a constant expression (e.g., the size of an array in a declaration).

Expression too complicated

The compiler has run out of stack space or expression blocks. Use multiple statements and temporary variables to evaluate the expression.

File type mismatch

The file appearing in this INSERT statement is not a text file.

Floating point not allowed in data list

A floating point number appears in a fixed point DATA statement or a floating point number appears in a constant expression that must evaluate to a fixed point value (e.g., the size of an array in a declaration).

Improper recursion

An attempt has been made to call this procedure recursively before all its formal parameters have been declared. Move all formal parameter declarations to the top of the procedure.

Improper type declaration

The RETURNS type of this procedure definition or forward reference is invalid or doesn't match this procedure's forward reference, or a storage class appears incorrectly (e.g., an automatic DATA statement, AUTOMATIC appearing outside a procedure, or formal parameters declared to be public or external, or formal parameters declared to be static in a recursive procedure or automatic in a non-recursive procedure).

Incorrect format

This line contains a syntax error.

Incorrect format in number

A character other than zero through seven appears in an octal constant or a character other than a digit or A through F appears in a hexadecimal constant.

Missing apostrophe

This line is missing the final apostrophe for a string constant.

Missing END statement

A module, procedure, or compound statement is missing an END statement. This is often caused by changing the simple statement of the THEN clause of an IF statement to a compound statement and forgetting to add its corresponding END.

Missing semicolon or format error

A semicolon was not found at the end of the last statement or this line contains a syntax error.

Missing subscript for '<identifier>'

An array reference is missing its subscript. This is often caused by passing an array as an actual parameter to a procedure that expects a scalar value.

'MODULE' must be the first statement of a library

A statement appears before this MODULE statement. Statements cannot appear outside a module.

Module name is missing

This MODULE statement does not include the name of the module.

Multiple 'MODULE' statements not allowed

This file already contains a module statement. Nesting modules or having more than one module in a file is not allowed.

Neither public nor swapping procs can be nested

This public or swapping procedure is defined within another procedure. Public and swapping procedures must be defined at the outermost layer (i.e., not within another procedure).

Nested or non-terminated comment

An attempt has been made to put a comment within a comment or a comment started some lines ago is missing its termination sequence */.

No treename specified

The INSERT, LIBRARY, or ENTER statement at this line does not include a treename.